Опубликовано в Научно-техническом вестнике СПбГУ ИТМО. 2008. Выпуск 53. Автоматное программирование, с. 145-162.

УДК 004.4'242

ВЕРИФИКАЦИЯ АВТОМАТНЫХ ПРОГРАММ С ИСПОЛЬЗОВАНИЕМ ВЕРИФИКАТОРА *SPIN*

М.А. Лукин, А.А. Шалыто
(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

Рассматривается способ использования верификатора *SPIN* для верификации автоматных программ. При использовании этого верификатора модель описывается на языке *Promela*, а требования – на языке *LTL*. В работе предлагается метод автоматизированного преобразования автоматных программ в модели на языке *Promela*. Для преобразования формулы на языке *LTL* в вид, пригодный для использования верификатором, разработано специальное программное средство. Использование предлагаемого метода иллюстрируется на примере верификации модели банкомата.
Ключевые слова: верификация программ, верификация на моделях, автоматное программирование, верификатор SPIN

Введение

В статье описывается методика автоматической верификации визуальных [9] автоматных программ [10]. Для поддержки этой методики разработан инструмент, позволяющий производить верификацию автоматных программ, спроектированных и реализованных на инструментальном средстве *UniMod* [11]. В данной работе верификация производится на основе метода *Model Checking* [12–18]. По данной теме уже проводились исследования [18–20], в которых модель программы строилась вручную. Такой подход имеет следующие недостатки:

- не гарантируется отсутствие ошибок в программе. Гарантируется лишь правильность алгоритма, на основе которого написана программа;
- модель приходится строить вручную, а это трудоемкий процесс.

Отличие предлагаемой методики заключается в том, что верификация программ автоматизирована, что позволяет избежать этих недостатков.

Верификация модели программы — это один из основных методов доказательства правильности программы. Для проведения верификации программы требуется произвести следующие действия.

1. Построить формальную модель, приемлемую для инструментальных средств верификации моделей. Обычно при построении модели абстрагируются от несущественных деталей программы в целях упрощения модели. При этом важно чтобы модель была адекватна программе. В общем случае модель строится вручную. В настоящей работе показано, что для автоматных программ этот этап можно выполнить автоматически.
2. Сформулировать требования к модели – составить спецификацию для модели. Обычно их формулируют на языке темпоральной логики. В настоящей работе требования формулируются на языке *LTL* [15–19, 21]. Эти требования для программ общего вида записать весьма сложно. Для автоматных программ эта задача решается проще.
3. Произвести верификацию модели. Для этого на вход верификатор подаются модель программы и требования к модели. Если модель не соответствует требованиям, пользователь получает трассу ошибки (контрпример), которая может помочь найти ошибку в программе. Иногда ошибка происходит в результате некорректного задания модели или неправильной спецификации (требований). В этом случае трасса ошибки помогает устранить ошибку в моделировании или спецификации. Для про-

грамм общего вида переход от контрпримера в терминах модели программы к контрпримеру в терминах самой программы весьма трудоемок. Для автоматных программ он может выполняться автоматически.

Существует большое число верификаторов, многие из которых имеют открытые исходные коды [22]. Одним из наиболее распространенных верификаторов является верификатор *SPIN* [15–17], который будет использоваться в настоящей работе. При его использовании модель описывается на языке *Promela* [15–17], а требования – на языке *LTL*.

Цель настоящей работы – повышение эффективности использования выбранного верификатора применительно к автоматным программам за счет автоматизации построения модели по программе и перехода от контрпримера к программе.

Синтаксис языка Promela

Синтаксис языка *Promela* напоминает синтаксис языка *C*. Модель на языке *Promela* состоит из следующих элементов: объявления типов данных; объявления каналов передачи переменных; объявления переменных; объявления и определения процессов; процесса `init`.

Понятие процесс в данном случае отдаленно можно рассматривать как процедуру, выполняемую в отдельном потоке. Приведем пример объявления и определения процесса:

```
proctype proc(int a; int b) {
byte c; /* локальная переменная */
/* тело процесса */
}
```

Процессы могут иметь параметры и локальные переменные. Процесс может быть запущен в нескольких экземплярах, если для него используется модификатор `active`. Запускаются процессы с помощью модификатора `run`.

Язык *Promela* имеет пять базовых типов данных: `bit`, `bool`, `byte`, `short`, `int`.

Тело процесса является последовательностью операторов. Операторы могут быть *выполнимыми* либо *заблокированными*. *Выполнимый* оператор может быть выполнен *немедленно*. *Заблокированный* оператор – оператор, который не может быть выполнен в данный момент. Такой оператор блокирует выполнение процесса до тех пор, пока он не станет *выполнимым*. Например, оператор

```
x < 7
```

может быть выполнен только когда *x* меньше семи. В противном случае он останавливает выполнение процесса до тех пор, пока условие не выполнится. Некоторые операторы, например, оператор присваивания, *выполнимы* всегда.

Язык *Promela* содержит также операторы ветвления и цикла, синтаксис (табл. 1) которых основан на охраняемых командах Дейкстры [23].

Таблица 1. Операторы ветвления и цикла

<table>
<tr><td><pre>if
   ::guard1 -> S1
   ::guard2 -> S2
   ...
   :: else -> Sk
fi</pre></td><td><pre>do
   ::guard1 -> S1
   ::guard2 -> S2
   ...
   :: else -> Sk
od</pre></td></tr>
</table>

Поясним работу операторов. Если условие *guardi* истинно, то выполняется действие *Si*. Если одновременно выполняются несколько условий, то происходит недетерминированный выбор одного из них. Если все условия ложны, то выполняется действие

Sk, соответствующее *else*. Конструкция *else* может отсутствовать. Тогда в случае, если все условия ложны, выполнение процесса блокируется до тех пор, пока хотя бы одно из них не начнет выполняться.

Модель Крипке

Пусть *AP* — множество атомарных высказываний. Модель *Крипке* [12, 13] над этим множеством – это четверка $M = (S, S0, R, L)$:

- S – конечное множество состояний;
- $S0 \subseteq S$ – множество начальных состояний;
- $R \subseteq S \times S$ – отношение переходов;
- $L : S \to 2AP$ – функция истинности.

Язык LTL

Для записи требований к модели используются языки темпоральной логики, например, *LTL, CTL, CTL**. Как отмечено выше, в настоящей работе используется язык *LTL* (*Linear-Time Logic*). Этот язык состоит из множества атомарных высказываний $p1, p2, \ldots \in AP$, логических операций ($\neg$, $\wedge$, $\vee$, $\to$) и темпоральных операторов. Пусть φ – правильно построенная формула. Тогда темпоральные операторы

- $X\varphi$ (в следующем состоянии φ верно – neXt);
- $G\varphi$ (φ верно всегда – Globally);
- $F\varphi$ (φ когда-нибудь будет верно – Finally);
- $\psi U\varphi$ (ψ будет верно до тех пор, пока не станет верно φ – Until)

– тоже правильно построенные формулы. Темпоральные операторы *G* и *F*, нужны для упрощения формул, их можно выразить через *U*:

- $F\varphi \equiv 1U\varphi$;
- $G\varphi \equiv \neg F \neg \varphi$.

Формулы *LTL* интерпретируются через исполнение системы переходов в модели *Крипке*. Если все пути из начального состояния удовлетворяют формуле φ, будем говорить, что поведение системы удовлетворяет формуле φ. В табл. 2 приведено соответствие стандартного синтаксиса и записи, принятой в верификаторе *SPIN*. Отметим, что *SPIN* не поддерживает оператор *X* (neXt), так как *SPIN* генерирует вспомогательные и служебные состояния в модели *Крипке*.

Таблица 2. Соответствие стандартного синтаксиса
и записи, принятой в верификаторе *r*

[]	G
<>	F
!	¬
U	U
&& или /\	/\
|| или \/	\/
->	→
<->	↔

Автомат Бюхи

Пусть *AP* – множество атомарных высказываний. Автоматом *Бюхи* [23–26] над алфавитом 2*AP* называется четверка $A = (Q, q0, \delta, F)$, где

- Q – конечное множество состояний;
- $q0$ – начальное состояние;
- $\delta \subseteq Q \times 2AP \times Q$ – функция переходов;
- $F \subseteq Q$ – множество допустимых состояний.

Доказано, что для любой *LTL*-формулы можно построить автомат *Бюхи*, который ее выполняет [26]. Более того, он может строиться автоматически. Например, рассмотрим *LTL*-формулу *G(p U q)* (В нотации *SPIN* эта формула записывается `[](p U q)`). Она обозначает, что всегда гарантировано, что *p* остается верным, по крайней мере, до тех пор, пока не станет верным *q*. *SPIN* ее транслирует в конструкцию *never claim:*

```
never { /* [](p U q) */
T0_init:
  if
  :: ((q)) -> goto accept_S9
  :: ((p)) -> goto T0_init
  fi;
accept_S9:
  if
  :: (((p)) || ((q)))) -> goto T0_init
  fi;
}
```

Эта конструкция соответствует автомату Бюхи, изображенному на рис. 1. Двойная линия обозначает допустимое состояние.

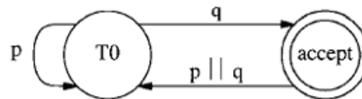

Рис. 1. Автомат Бюхи для формулы *G(p U q)*

С помощью автомата *Бюхи* можно верифицировать модель *Крипке*. С точки зрения верификации автоматных моделей – это наиболее удобный вариант, позволяющий при верификации и спецификации почти полностью ограничиться понятием *конечный автомат*.

Построение модели на языке *Promela*

В данном разделе приводится алгоритм построения модели на языке *Promela* по автоматной программе.

1. Подготовка исходных данных.
   1. Для каждого автомата `Ai` заведем переменную `stateAi`, в которой будет храниться номер текущего состояния. На языке *Promela* это описывается так:
      `int stateAi;`
   2. Заведем одну переменную для событий:
      `int lastEvent;`
   3. Каждому состоянию присвоим уникальный номер, используя сквозную нумерацию для всех автоматов. Переход в новое состояние с номером `k` будет осуществляться присвоением переменной `stateAi` числа `k`:
      `stateAi = k;`
   4. Событие `exx` (`xx` — номер события) на переходе между состояниями описывается в модели следующим кодом:
      `lastEvent = xx;`
2. Построение
   5. Для каждого автомата `Ai` создадим функцию. На языке *Promela* это записывается следующим образом:

```
inline Ai() {
```

```
/* тело функции */
}
```
6. Для каждого автомата `Ai` в теле созданной функции выполнить шаги 2.3 – 2.9.
7. Определить начальное (стартовое) состояние `s`. Присвоить:
```
stateAi = s;
```
8. Построить цикл:
```
do
::(stateAi == s1) ->
  printf("State Ai 1 : Имя состояния\n");
::(stateAi == s2) ->
  printf("State Ai 2 : Имя состояния\n");
…
::(stateAi == sk) ->
  printf("State Ai k : Имя состояния\n");
od;
```
Здесь `s1`, …, `sk` – номера состояний автомата `Ai`. С помощью инструкции `printf` введена пометка о том, что автомат вошел в данное состояние. Инструкция `printf` аналогична соответствующей инструкции из языка *C*. Пометка используется в дальнейшем для восстановления контрпримера.

9. Если в некоторое состояние `sj` вложен автомат `Am`, то дописать в условие `(stateAi == sj)` пометку о вложенном автомате и вызов функции этого автомата:
```
printf("Calling automaton Am\n");
Am();
```
10. Для каждого состояния `sj` найти все возможные переходы `(sj, sl)` из него. К условию `(stateAi == sj)` дописать конструкцию `if`, а для каждого перехода `(sj, sl)` дописать в конструкции `if` следующее:
```
::stateAi = sl;
 printf("Going to state Ai sl : Имя состояния\n");
```
Это обозначает, что для присваивания `stateAi = sl` необходимое условие выполнено всегда. Также добавляется пометка о переходе в новое состояние.

В результате получится конструкция следующего вида:
```
if
::stateAi = s1
 printf("Going to state Ai s1 : Имя состояния\n");
::stateAi = s2
 printf("Going to state Ai s2 : Имя состояния\n");
…
::stateAi = sk
 printf("Going to state Ai sk : Имя состояния\n");
fi;
```
Эта конструкция обозначает, что присваивание нового номера состояния происходит недетерминировано. Таким образом, верификатор проверит все варианты переходов в новое состояние.

11. Если переход помечен событием `exx`, то дописать выражение
```
lastEvent = xx;
```
и пометку
```
printf("Event = exx\n");
```
Таким образом, в результате получается конструкция вида:
```
if
…
::stateAi = sl;
 printf("Going to state Ai sl : Имя состояния\n");
 lastEvent = xx;
 printf("Event = exx\n");
```

```
...
fi;
```
12. Если состояние `st` – конечное, то в условие `(stateAi == st)` дописать инструкцию завершения цикла:
```
break;
```
13. После построения функций для всех автоматов определить стартовый автомат `Ai` и создать процесс, его запускающий. На языке *Promela* это записывается следующим образом:
```
proctype Model() {
  Ai();
}

init {
run Model();
}
```
14. Дописать в модель требования (проверяемые свойства), преобразованные с помощью верификатора *SPIN* с языка *LTL* на язык *Promela*.

Например, рассмотрим автомат из программы [27] (автомат *A3*, рис. 2) и его модель на языке *Promela*, сгенерированную по автомату на основе предлагаемого метода построения модели. В пример не входит построение требований. В данном примере строится модель на языке *Promela* для автомата как для целой программы для того, чтобы продемонстрировать выполнение шага 2.9 алгоритма.

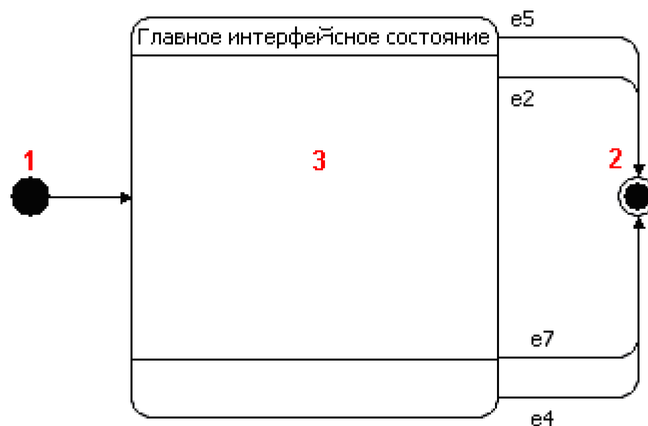

Рис. 2. Автомат *A3*

В автомате *A3* три состояния: `Начальное`, `Конечное` и `Главное интерфейсное состояние`. Перенумеруем их как показано на рис. 2. Модель автомата *A3*, построенная по излагаемой методике, выглядит следующим образом:
```
int stateA3;
inline A3() {
  stateA3 = 1;
  do
  ::(stateA3 == 3) ->
    printf("State A3 3 : Главное интерфейсное состояние\n");
    if
    ::stateA3 = 2;
     printf("Going to state A3 2 : Конечное \n");
     lastEvent = 4;
     printf("Event = e4\n");
    ::stateA3 = 2;
     printf("Going to state A3 2 : Конечное \n");
```

```
         lastEvent = 7;
         printf("Event = e7\n");
        ::stateA3 = 2;
         printf("Going to state A3 2 : Конечное \n");
         lastEvent = 5;
         printf("Event = e5\n");
        ::stateA3 = 2;
         printf("Going to state A3 2 : Конечное \n");
         lastEvent = 2;
         printf("Event = e5\n");
        fi;
  ::(stateA3 == 1) ->
        printf("State A3 1 : Начальное \n");
        if
        ::stateA3 = 3;
         printf("Going to state A3 3 : Главное интерфейсное
             состояние\n");
        fi;
  ::(stateA3 == 2) ->
        printf("State A3 2 : Конечное \n");
        break;
  od;
}
proctype Model() {
  A3();
}
init {
  run Model();
}
```

Преобразование LTL-формул

Каждому элементарному высказыванию присвоим идентификатор *pk*. В модель запишем следующий код:

```
#define pk <элементарное_высказывание>
```

После этого заменим все элементарные высказывания их идентификаторами. В таком виде *LTL*-формула может быть обработана верификатором *SPIN* (при условии, что темпоральные операторы записаны в нотации *SPIN*).

Построение контрпримера

Верификатор *SPIN* автоматически строит контрпример как путь в модели, поданной ему на вход. Так как модель на языке *Promela* эквивалентна исходному автомату, обратное преобразование не требуется.

Общее описание инструментального средства *Converter*

Это инструментальное средство является практической реализацией предложенной методики верификации автоматных программ и позволяет производить полностью автоматическую верификацию автоматных программ. По автоматной программе *Converter* создает модель, в которой отброшены несущественные детали. *LTL*-формула преобразовывается в пригодный для верификатора SPIN вид. Инструментальное средство принимает на вход три параметра: путь к *XML*-описанию автоматной программы, имя файла, в который записывается созданная модель и *LTL*-формулу со спецификацией. На выходе *Converter* выдает файл *report.txt*, в котором записан полный отчет о верификации.

Описание работы инструментального средства

Автоматическая верификация автоматных программ состоит из нескольких этапов (рис. 3).

Рис. 3. Общая схема верификации

3. *Converter* принимает на вход визуальную автоматную программу, разработанную при помощи инструментального средства *UniMod* и сохраненную в формате *XML*, и требования к ней, записанные на языке *LTL* в расширенной нотации (рис. 3, переход 1 – 2). *Важно: верификатору на вход подаются не требования, а их отрицание.*
4. При помощи *UniMod* из *XML*-файла получается автоматная модель на языке *Java* (рис. 4) – переход 1-3 на рис. 3.

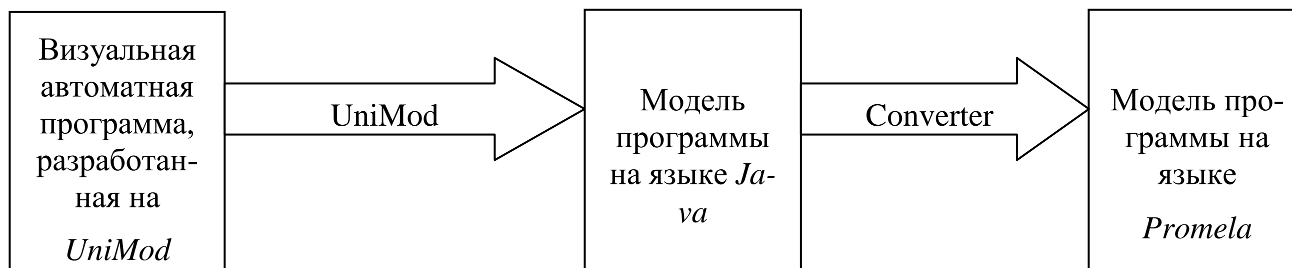

Рис. 4. Взаимодействие с *UniMod*

5. *Converter* транслирует автоматную модель с языка Java на язык *Promela* (рис. 4) – переход 1-3 на рис. 3.
6. *Converter* преобразует *LTL*-формулу в нотацию верификатора *SPIN* (рис. 5). Это преобразование выполняется следующим образом (это также переход 2-3 на рис. 3).

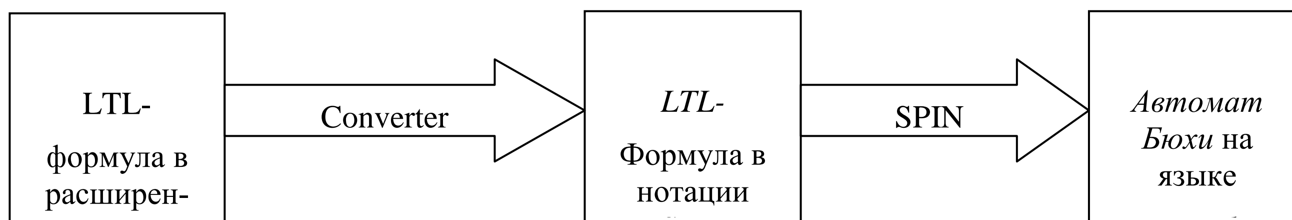

Рис. 5. Преобразование *LTL*-формул

При этом должны выполняться следующие условия.

- Все элементарные высказывания должны быть записаны в фигурных скобках.
- Все элементарные высказывания должны удовлетворять синтаксису языка *Promela*.
- Каждому элементарному высказыванию присваивается идентификатор pk, где *k* — порядковый номер элементарного высказывания в формуле.
- Для каждого элементарного высказывания *Converter* генерирует макрос на языке *Promela*, закрепляющий идентификатор за элементарным высказыванием. Если *Converter* распознает формулу как неправильную, то он выводит в консоль сообщение «Wrong formula».
- Идентификаторы событий exx, где *xx* – номер события, преобразовываются в число *xx*.
- Пример. Формула (элементарное высказывание)

```
{lastEvent == e13}
```

преобразовывается в строку на *Promela*:

```
#define pk (lastEvent == 13),
```

где *k* — номер элементарного высказывания.

7. *Converter* запускает верификатор *SPIN* в режиме генерации конструкции *never claim* (с помощью команды `spin -f <формула>`). Верификатор по *LTL*-формуле генерирует конструкцию n*ever claim*, представляющую собой *автомат Бюхи*, записанный на языке *Promela* (рис. 5). Это также переход 2–3 на рис. 3.
8. После того, как на языке *Promela* описаны модель и требование к ней, *Converter* запускает *SPIN* на верификацию (командой `spin -a <Модель>`). Верификатор *SPIN* генерирует файл `pan.c`, представляющий собой программу-верификатор на языке *C* (переход 3–4 на рис. 3).
9. После компиляции файла `pan.c` получается программа-верификатор для данной конкретной модели с заданным требованием. Программа *pan* выполняет верификацию построенной модели. При обнаружении ошибок программа *pan* выдает *trail*-файл, в котором описана трасса ошибки в формате, понятном верификатору *SPIN* (переходы 4–6 на рис. 3). Кроме того, программа-верификатор *pan* выводит отчет, в котором содержится краткая информация об ошибках (переход 5–6 на рис. 3), использованной памяти, версия *SPIN* и т.д.
10. По команде `spin -t -p <Модель>` верификатор выводит отчет, содержащий контрпример. *Converter* собирает воедино отчет, созданный *SPIN*, и отчет, созданный программой *pan* (переход 6–7 на рис. 3).

Обратное преобразование контрпримера

Верификатор *SPIN* выдает контрпример в виде пути в модели, описанной на языке *Promela*. Благодаря системе пометок контрпример автоматически преобразовывается в удобный для восприятия текст. Отчет, составленный инструментом *Converter*, содержит в себе оба варианта. Строка отчета

```
Calling automaton Ai
```

обозначает вход в автомат *Ai*.

Строка отчета

```
Going to state Ai s : Имя состояния
```

обозначает переход автомата *Ai* в состояние *s*.

Строка отчета

```
Event = exx
```

обозначает, что на данном переходе есть событие *exx*.

Строка отчета

```
State Ai s : Имя состояния
```

обозначает, что автомат *Ai* попал в состояние *s*.

Строка отчета

```
lastEvent = exx,
```

где *exx* – некоторое событие обозначает, что был совершен переход по событию *exx*.

Таким образом строится путь в автоматной модели из файла отчета, выданного инструментом *Converter*.

Пример использования автоматической верификации

Постановка задачи

Проведем верификацию программной модели банкомата [20], разработанной при помощи инструментального средства *UniMod*. На рис. 6, 7 показаны номера, которые *Converter* присвоил состояниям системы автоматов.

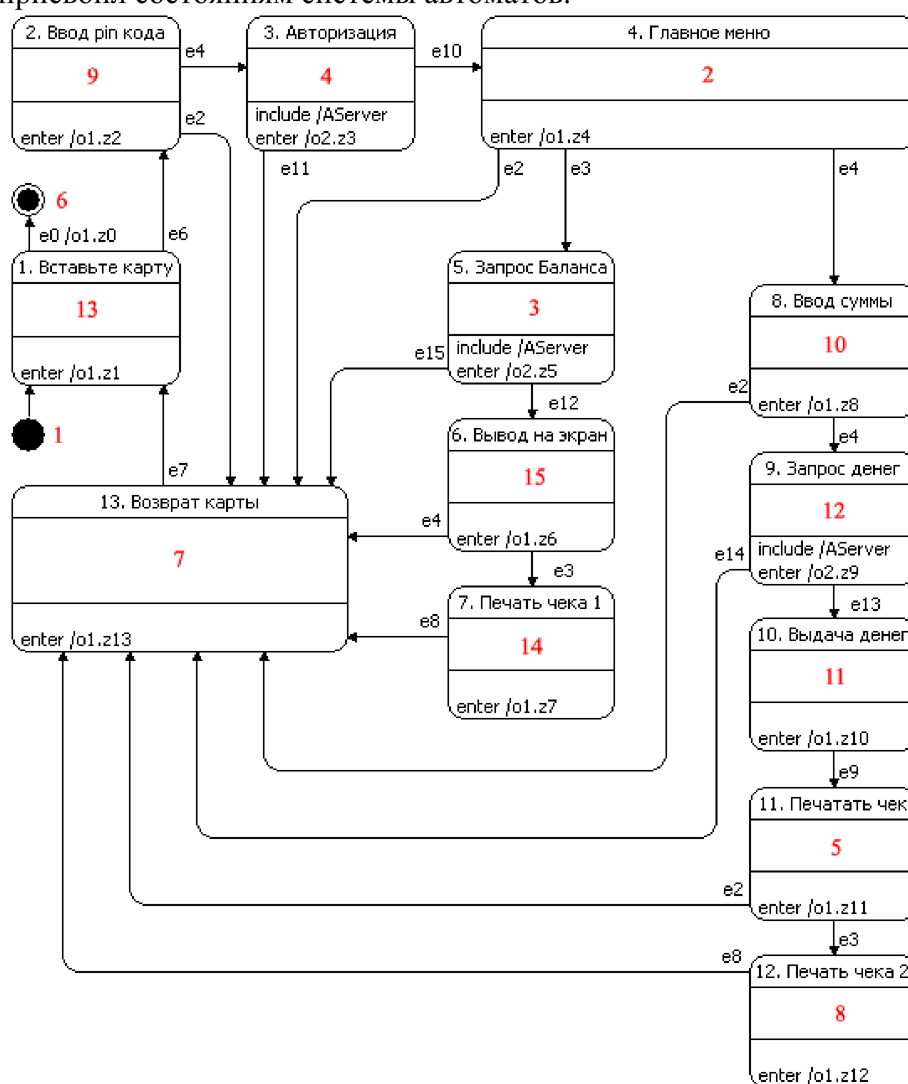

Рис. 6. Автоматная реализация банкомата. Автомат *AClient*

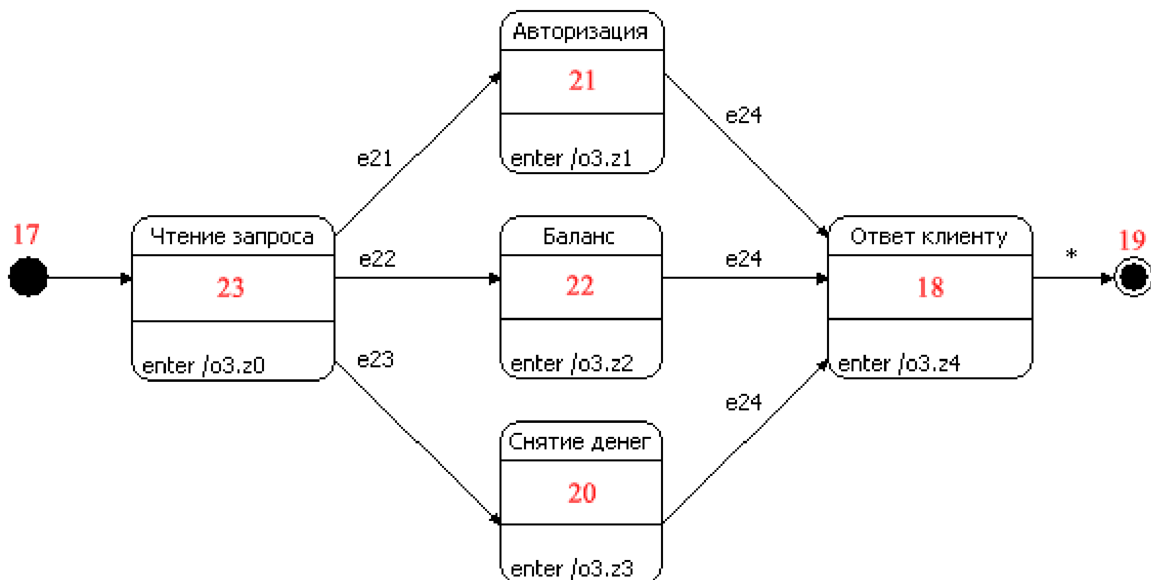

Рис. 7. Автоматная реализация банкомата. Автомат *AServer*

Верификация

Проверим, выдает ли банкомат деньги. Для этого проверим свойство «Банкомат не дает денег». Напомним, что на вход верификатору *SPIN* требуется подавать отрицания проверяемых свойств. Запишем отрицание этого свойства на русском языке: «Когда-нибудь банкомат выдаст деньги». Для построения формулы на языке *LTL*, соответствующей данному свойству, введем атомарное высказывание: `stateAClient == 11`. Оно обозначает, что автомат *AClient* попал в состояние №11 «10. Выдача денег» и выдал деньги. Перефразируем данное свойство (точнее, его отрицание) через введенное элементарное высказывание: «Автомат *AClient* попадет в состояние «10. Выдача денег»». Следовательно, *LTL*-формула для этого свойства будет выглядеть следующим образом: `<>({stateAClient == 11})`.

Ожидаемый результат: в отчете о проведенной верификации должно быть сообщение об ошибке и выведенный контрпример. В этом контрпримере должен быть путь по состояниям автомата *AClient* до состояния «*10. Выдача денег*». Проведем верификацию. Инструментальное средство *Converter* выдаст следующий отчет:

```
Converter v. 0.50
warning: for p.o. reduction to be valid the never claim must be
          stutter-invariant
(never claims generated from LTL formulae are stutter-
          invariant)
pan: claim violated! (at depth 173)
pan: wrote models/aclient.ltl.trail
(Spin Version 4.2.8 -- 6 January 2007)
Warning: Search not completed
   + Partial Order Reduction

Full statespace search for:
   never claim    +
   assertion violations  + (if within scope of claim)
   acceptance cycles  - (not selected)
   invalid end states - (disabled by never claim)

State-vector 28 byte, depth reached 187, errors: 1
 198 states, stored
 10 states, matched
 208 transitions (= stored+matched)
```

```
 0 atomic steps
hash conflicts: 0 (resolved)

2.622  memory usage (Mbyte)

Starting :init: with pid 0
Starting :never: with pid 1
Never claim moves to line 267 [(1)]
Starting Model with pid 2
 2:  proc 0 (:init:) line 261 "models/aclient.ltl" (state 1)
            [(run Model())]
 4:  proc 1 (Model) line 23 "models/aclient.ltl" (state 1)
            [stateAClient = 1]
 6:  proc 1 (Model) line 25 "models/aclient.ltl" (state 2)
            [((stateAClient==1))]
 State AClient 1 : s1
 8:  proc 1 (Model) line 26 "models/aclient.ltl" (state 3)
            [printf('State AClient 1 : s1\\n')]
 10: proc 1 (Model) line 28 "models/aclient.ltl" (state 4)
            [stateAClient = 13]
 Going to state AClient 13 : 1. Вставьте карту
 12: proc 1 (Model) line 29 "models/aclient.ltl" (state 5)
            [printf('Going to state AClient 13 : 1. Вставьте
         карту\\n')]
 14: proc 1 (Model) line 152 "models/aclient.ltl" (state 306)
            [((stateAClient==13))]
 State AClient 13 : 1. Вставьте карту
 16: proc 1 (Model) line 153 "models/aclient.ltl" (state 307)
            [printf('State AClient 13 : 1. Вставьте карту\\n')]
 18: proc 1 (Model) line 159 "models/aclient.ltl" (state 312)
            [stateAClient = 9]
 Going to state AClient 9 : 2. Ввод pin кода
 20: proc 1 (Model) line 160 "models/aclient.ltl" (state 313)
            [printf('Going to state AClient 9 : 2. Ввод pin
         кода\\n')]
 22: proc 1 (Model) line 161 "models/aclient.ltl" (state 314)
            [lastEvent = 6]
 Event = e6
 24: proc 1 (Model) line 162 "models/aclient.ltl" (state 315)
            [printf('Event = e6\\n')]
 26: proc 1 (Model) line 106 "models/aclient.ltl" (state 201)
            [((stateAClient==9))]
 State AClient 9 : 2. Ввод pin кода
 28: proc 1 (Model) line 107 "models/aclient.ltl" (state 202)
            [printf('State AClient 9 : 2. Ввод pin кода\\n')]
 30: proc 1 (Model) line 109 "models/aclient.ltl" (state 203)
            [stateAClient = 4]
 Going to state AClient 4 : 3. Авторизация
 32: proc 1 (Model) line 110 "models/aclient.ltl" (state 204)
            [printf('Going to state AClient 4 : 3.
         Авторизация\\n')]
 34: proc 1 (Model) line 111 "models/aclient.ltl" (state 205)
            [lastEvent = 4]
 Event = e4
 36: proc 1 (Model) line 112 "models/aclient.ltl" (state 206)
            [printf('Event = e4\\n')]
 38: proc 1 (Model) line 61 "models/aclient.ltl" (state 97)
            [((stateAClient==4))]
 State AClient 4 : 3. Авторизация
 40: proc 1 (Model) line 62 "models/aclient.ltl" (state 98)
            [printf('State AClient 4 : 3. Авторизация\\n')]
```

```
Calling automaton AServer
40: proc 1 (Model) line 63 "models/aclient.ltl" (state 99)
          [printf('Calling automaton AServer\\n')]
42: proc 1 (Model) line 197 "models/aclient.ltl" (state 100)
          [stateAServer = 17]
44: proc 1 (Model) line 199 "models/aclient.ltl" (state 101)
          [((stateAServer==17))]
State AServer 17 : s1
46: proc 1 (Model) line 200 "models/aclient.ltl" (state 102)
          [printf('State AServer 17 : s1\\n')]
48: proc 1 (Model) line 202 "models/aclient.ltl" (state 103)
          [stateAServer = 23]
Going to state AServer 23 : Чтение запроса
50: proc 1 (Model) line 203 "models/aclient.ltl" (state 104)
          [printf('Going to state AServer 23 : Чтение
        запроса\\n')]
52: proc 1 (Model) line 238 "models/aclient.ltl" (state 140)
          [((stateAServer==23))]
State AServer 23 : Чтение запроса
54: proc 1 (Model) line 239 "models/aclient.ltl" (state 141)
          [printf('State AServer 23 : Чтение запроса\\n')]
56: proc 1 (Model) line 241 "models/aclient.ltl" (state 142)
          [stateAServer = 20]
Going to state AServer 20 : Снятие денег
58: proc 1 (Model) line 242 "models/aclient.ltl" (state 143)
          [printf('Going to state AServer 20 : Снятие
        денег\\n')]
60: proc 1 (Model) line 243 "models/aclient.ltl" (state 144)
          [lastEvent = 23]
Event = e23
62: proc 1 (Model) line 244 "models/aclient.ltl" (state 145)
          [printf('Event = e23\\n')]
64: proc 1 (Model) line 214 "models/aclient.ltl" (state 116)
          [((stateAServer==20))]
State AServer 20 : Снятие денег
66: proc 1 (Model) line 215 "models/aclient.ltl" (state 117)
          [printf('State AServer 20 : Снятие денег\\n')]
68: proc 1 (Model) line 217 "models/aclient.ltl" (state 118)
          [stateAServer = 18]
Going to state AServer 18 : Ответ клиенту
70: proc 1 (Model) line 218 "models/aclient.ltl" (state 119)
          [printf('Going to state AServer 18 : Ответ
        клиенту\\n')]
72: proc 1 (Model) line 219 "models/aclient.ltl" (state 120)
          [lastEvent = 24]
Event = e24
74: proc 1 (Model) line 220 "models/aclient.ltl" (state 121)
          [printf('Event = e24\\n')]
76: proc 1 (Model) line 205 "models/aclient.ltl" (state 107)
          [((stateAServer==18))]
State AServer 18 : Ответ клиенту
78: proc 1 (Model) line 206 "models/aclient.ltl" (state 108)
          [printf('State AServer 18 : Ответ клиенту\\n')]
80: proc 1 (Model) line 208 "models/aclient.ltl" (state 109)
          [stateAServer = 19]
Going to state AServer 19 : s2
82: proc 1 (Model) line 209 "models/aclient.ltl" (state 110)
          [printf('Going to state AServer 19 : s2\\n')]
84: proc 1 (Model) line 211 "models/aclient.ltl" (state 113)
          [((stateAServer==19))]
State AServer 19 : s2
```

```
 86: proc 1 (Model) line 212 "models/aclient.ltl" (state 114)
            [printf('State AServer 19 : s2\\n')]
 88: proc 1 (Model) line 66 "models/aclient.ltl" (state 160)
            [stateAClient = 2]
 Going to state AClient 2 : 4. Главное меню
 90: proc 1 (Model) line 67 "models/aclient.ltl" (state 161)
            [printf('Going to state AClient 2 : 4. Главное
         меню\\n')]
 92: proc 1 (Model) line 68 "models/aclient.ltl" (state 162)
            [lastEvent = 10]
 Event = e10
 94: proc 1 (Model) line 69 "models/aclient.ltl" (state 163)
            [printf('Event = e10\\n')]
 96: proc 1 (Model) line 31 "models/aclient.ltl" (state 8)
            [((stateAClient==2))]
 State AClient 2 : 4. Главное меню
 98: proc 1 (Model) line 32 "models/aclient.ltl" (state 9)
            [printf('State AClient 2 : 4. Главное меню\\n')]
100: proc 1 (Model) line 42 "models/aclient.ltl" (state 18)
            [stateAClient = 10]
 Going to state AClient 10 : 8. Ввод суммы
102: proc 1 (Model) line 43 "models/aclient.ltl" (state 19)
            [printf('Going to state AClient 10 : 8. Ввод
         суммы\\n')]
104: proc 1 (Model) line 44 "models/aclient.ltl" (state 20)
            [lastEvent = 4]
 Event = e4
106: proc 1 (Model) line 45 "models/aclient.ltl" (state 21)
            [printf('Event = e4\\n')]
108: proc 1 (Model) line 118 "models/aclient.ltl" (state 213)
            [((stateAClient==10))]
 State AClient 10 : 8. Ввод суммы
110: proc 1 (Model) line 119 "models/aclient.ltl" (state 214)
            [printf('State AClient 10 : 8. Ввод суммы\\n')]
112: proc 1 (Model) line 121 "models/aclient.ltl" (state 215)
            [stateAClient = 12]
 Going to state AClient 12 : 9. Запрос денег
114: proc 1 (Model) line 122 "models/aclient.ltl" (state 216)
            [printf('Going to state AClient 12 : 9. Запрос
         денег\\n')]
116: proc 1 (Model) line 123 "models/aclient.ltl" (state 217)
            [lastEvent = 4]
 Event = e4
118: proc 1 (Model) line 124 "models/aclient.ltl" (state 218)
            [printf('Event = e4\\n')]
120: proc 1 (Model) line 138 "models/aclient.ltl" (state 233)
            [((stateAClient==12))]
 State AClient 12 : 9. Запрос денег
122: proc 1 (Model) line 139 "models/aclient.ltl" (state 234)
            [printf('State AClient 12 : 9. Запрос денег\\n')]
 Calling automaton AServer
122: proc 1 (Model) line 140 "models/aclient.ltl" (state 235)
            [printf('Calling automaton AServer\\n')]
124: proc 1 (Model) line 197 "models/aclient.ltl" (state 236)
            [stateAServer = 17]
126: proc 1 (Model) line 199 "models/aclient.ltl" (state 237)
            [((stateAServer==17))]
 State AServer 17 : s1
128: proc 1 (Model) line 200 "models/aclient.ltl" (state 238)
            [printf('State AServer 17 : s1\\n')]
```

```
130: proc 1 (Model) line 202 "models/aclient.ltl" (state 239)
            [stateAServer = 23]
 Going to state AServer 23 : Чтение запроса
132: proc 1 (Model) line 203 "models/aclient.ltl" (state 240)
            [printf('Going to state AServer 23 : Чтение
          запроса\\n')]
134: proc 1 (Model) line 238 "models/aclient.ltl" (state 276)
            [((stateAServer==23))]
 State AServer 23 : Чтение запроса
136: proc 1 (Model) line 239 "models/aclient.ltl" (state 277)
            [printf('State AServer 23 : Чтение запроса\\n')]
138: proc 1 (Model) line 241 "models/aclient.ltl" (state 278)
            [stateAServer = 20]
 Going to state AServer 20 : Снятие денег
140: proc 1 (Model) line 242 "models/aclient.ltl" (state 279)
            [printf('Going to state AServer 20 : Снятие
          денег\\n')]
142: proc 1 (Model) line 243 "models/aclient.ltl" (state 280)
            [lastEvent = 23]
 Event = e23
144: proc 1 (Model) line 244 "models/aclient.ltl" (state 281)
            [printf('Event = e23\\n')]
146: proc 1 (Model) line 214 "models/aclient.ltl" (state 252)
            [((stateAServer==20))]
 State AServer 20 : Снятие денег
148: proc 1 (Model) line 215 "models/aclient.ltl" (state 253)
            [printf('State AServer 20 : Снятие денег\\n')]
150: proc 1 (Model) line 217 "models/aclient.ltl" (state 254)
            [stateAServer = 18]
 Going to state AServer 18 : Ответ клиенту
152: proc 1 (Model) line 218 "models/aclient.ltl" (state 255)
            [printf('Going to state AServer 18 : Ответ
          клиенту\\n')]
154: proc 1 (Model) line 219 "models/aclient.ltl" (state 256)
            [lastEvent = 24]
 Event = e24
156: proc 1 (Model) line 220 "models/aclient.ltl" (state 257)
            [printf('Event = e24\\n')]
158: proc 1 (Model) line 205 "models/aclient.ltl" (state 243)
            [((stateAServer==18))]
 State AServer 18 : Ответ клиенту
160: proc 1 (Model) line 206 "models/aclient.ltl" (state 244)
            [printf('State AServer 18 : Ответ клиенту\\n')]
162: proc 1 (Model) line 208 "models/aclient.ltl" (state 245)
            [stateAServer = 19]
 Going to state AServer 19 : s2
164: proc 1 (Model) line 209 "models/aclient.ltl" (state 246)
            [printf('Going to state AServer 19 : s2\\n')]
166: proc 1 (Model) line 211 "models/aclient.ltl" (state 249)
            [((stateAServer==19))]
 State AServer 19 : s2
168: proc 1 (Model) line 212 "models/aclient.ltl" (state 250)
            [printf('State AServer 19 : s2\\n')]
170: proc 1 (Model) line 143 "models/aclient.ltl" (state 296)
            [stateAClient = 11]
Never claim moves to line 266 [((stateAClient==11))]
 Going to state AClient 11 : 10. Выдача денег
172: proc 1 (Model) line 144 "models/aclient.ltl" (state 297)
            [printf('Going to state AClient 11 : 10. Выдача
          денег\\n')]
Never claim moves to line 270 [(1)]
```

```
spin: trail ends after 174 steps
#processes: 2
     lastEvent = 24
     stateAClient = 11
     stateAServer = 19
174: proc 1 (Model) line 145 "models/aclient.ltl" (state 298)
174: proc 0 (:init:) line 262 "models/aclient.ltl" (state 2)
          <valid end state>
174: proc - (:never:) line 271 "models/aclient.ltl" (state 8)
          <valid end state>
2 processes created
```

Строка отчета

```
State-vector 28 byte, depth reached 133, errors: 1
```

говорит о том, что была найдена ошибка. Выпишем контрпример в явном виде:

Автомат AClient перешел в состояние *s1*.
Автомат *AClient* перешел в состояние *1. Вставьте карту*.
Произошло событие *e6*.
Автомат *AClient* перешел в состояние *2. Ввод pin кода*.
Произошло событие *e4*.
Автомат *AClient* перешел в состояние *3. Авторизация*.
Автомат *AServer* перешел в состояние *s1*.
Автомат *AServer* перешел в состояние *Чтение запроса*.
Произошло событие *e23*.
Автомат *AServer* перешел в состояние *Снятие денег*.
Произошло событие *e24*.
Автомат *AServer* перешел в состояние *Ответ клиенту*.
Автомат *AServer* перешел в состояние *s2*.
Произошло событие *e10*.
Автомат *AClient* перешел в состояние *4. Главное меню*.
Произошло событие *e4*.
Автомат *AClient* перешел в состояние *8. Ввод суммы*.
Произошло событие *e4*.
Автомат *AClient* перешел в состояние *9. Запрос денег*.
Автомат *AServer* перешел в состояние *s1*.
Автомат *AServer* перешел в состояние *Чтение запроса*.
Произошло событие *e23*.
Автомат *AServer* перешел в состояние *Снятие денег*.
Произошло событие *e24*.
Автомат *AServer* перешел в состояние *Ответ клиенту*.
Автомат *AServer* перешел в состояние *s2*.
Произошло событие *e13*.
Автомат *AClient* перешел в состояние *10. Выдача денег*.

Как и ожидалось, контрпример содержит путь по состояниям автомата *AClient* до состояния «*10. Выдача денег*».

Заключение

В предложенном методе верификации автоматных программ используется верификатор *SPIN* – один из наиболее мощных и известных верификаторов. Его используют *NASA* [21] и многие другие организации, где требуется повышенная надежность.

Автоматные программы удобны для проектирования и наглядны. Кроме того, они позволяют автоматически построить модель Крипке и произвести верификацию.

Настоящая работа предлагает метод автоматической верификации автоматных программ, написанных в среде *UniMod* с помощью верификатора *SPIN*.

Литература

1. Новиков Ф.А. Визуальное конструирование программ. – Режим доступа: http://is.ifmo.ru/works/visualcons/
2. Шалыто А.А. Switch-технология. Алгоритмизация и программирование задач логического управления. – СПб: Наука, 1998. – Режим доступа: http://is.ifmo.ru/books/switch/1/
3. UniMod home page. – Режим доступа: http://UniMod.sourceforge.net/
4. Лифшиц Ю. Верификация программ и темпоральные логики. Лекция № 3 курса «Современные задачи теоретической информатики». – Режим доступа: http://download.yandex.ru/class/lifshits/lecture-note03.pdf
5. Лифшиц Ю. Символьная верификация программ. Лекция № 4 курса «Современные задачи теоретической информатики». – Режим доступа: http://download.yandex.ru/class/lifshits/lecture-note04.pdf
6. Кларк Э.М., Грамберг О., Пелед Д. Верификация моделей программ: Model Checking. – М.: МЦНМО, 2002.
7. Holzman G. J. Design And Validation Of Computer Protocols. – Prentice Hall, 1991.
8. Holzman G. J. The model checker SPIN //IEEE Trans. on Software Engineering. –1997. – №23. – P. 279–295.
9. SPIN home page. – Режим доступа: http://SPINroot.com
10. Вельдер С.Э., Шалыто А.А. О верификации простых автоматных программ на основе метода Model checking // Информационно-управляющие системы. – 2007. – № 3. – С. 27–38. – Режим доступа: http://is.ifmo.ru/download/27-38.pdf
11. Васильева К.А., Кузьмин Е.В. Верификация автоматных программ с использованием LTL // Моделирование и анализ информационных систем. – 2007. – № 1. – С.3–14. – Режим доступа: http://is.ifmo.ru/verification/_LTL_for_Spin.pdf
12. Виноградов Р.А., Кузьмин Е.В., Соколов В.А. Верификация автоматных программ средствами CPN/Tools / Доклады II-й научно-методической конференции преподавателей матем. ф-та и ф-та ИВТ Ярославского гос. ун-та им. П.Г. Демидова «Преподавание математики и компьютерных наук в классическом университете». – Ярославль: ЯрГУ. 2007. – С. 91–101. – Режим доступа: http://is.ifmo.ru/verification/_ver_prog.pdf
13. Linear temporal logic. – Режим доступа: http://en.wikipedia.org/wiki/Linear_temporal_logic
14. Белешко Д.С. Верификация автоматных моделей программ. Бакалаврская работа. – СПбГУ ИТМО, 2006.
15. Dijkstra E. W. Guarded commands, non-determinacy and formal derivation of programs // CACM. – 1975. – №8.
16. Büchi automaton. – Режим доступа: http://en.wikipedia.org/wiki/Büchi_automaton
17. Кюзара В.Е. Реализация системы проверки моделей раскрашенных сетей Петри с использованием разверток. – Режим доступа: www.iis.nsk.su/preprints/pdf/094.pdf
18. Vardy M., Wolper P. An automata-theoretic approach to automatic program verification // Proc. 1st IEEE Symp. On Logic in Computer Science. – 1986.
19. Яковлев А.В., Лукин М.А., Шалыто А.А. Реализация классической игры «Ним» на основе автоматного подхода. – СПбГУ ИТМО. 2006. – Режим доступа: http://is.ifmo.ru/UniMod-projects/knim/
20. Козлов В.А., Комалева О.А. Моделирование работы банкомата. – СПбГУ ИТМО. 2006. – Режим доступа: http://is.ifmo.ru/UniMod-projects/bankomat/
21. Риган П., Хемилтон С. NASA: миссия надежна // Открытые системы. – 2004. – № 3. – Режим доступа: http://www.osp.ru/os/2004/03/184060/